Title
Self-stabilizing distributed constraint satisfaction

Authors
Collin, Zeev
Dechter, Rina
Katz, Shmuel

Publication Date
1992

Peer reviewed

Self-Stabilizing Distributed Constraint Satisfaction

Zeev Collin
Computer Science Department
Technion, Haifa, Israel

Rina Dechter
Information and Computer Science
University of California, Irvine, CA

Shmuel Katz
Computer Science Department
Technion, Haifa, Israel

Technical Report 92-67

January, 1992

This work was partially supported by the National Science Foundation, Grant #IRI-8821444, the Air Force Office of Scientific Research, Grant #AFOSR-90-0136, GE Corporate R&D and by Toshiba of America.

Self-Stabilizing Distributed Constraint Satisfaction

Zeev Collin
Computer Science Department
Technion, Haifa, Israel

Rina Dechter*
Information and Computer Science
UCI, Irvine, CA

Shmuel Katz
Computer Science Department
Technion, Haifa, Israel

Abstract

This paper characterizes connectionist-type architectures that allow a distributed solution for classes of constraint satisfaction problems, and presents such solutions. We first consider whether there exists a **uniform** model of computation that guarantees convergence to a solution from every initial state of the system, whenever such a solution exists. Even for relatively simple constraint networks, such as rings, we show that there is no general solution that guarantees convergence from every initial state of the system using a completely uniform, asynchronous model. However, some restricted topologies such as trees can accommodate the uniform, asynchronous model and a protocol demonstrating this fact is presented. An **almost-uniform**, asynchronous, network consistency protocol is also presented. Subprotocols are given to make an undirected tree directed and to traverse a graph. We show that the algorithms are guaranteed to be self-stabilizing, which makes them suitable for dynamic or error-prone environments.

1 Introduction

Consider the distributed version of the graph coloring problem, where each node must select a color (from a given set of colors) that is different from any color selected by its neighbors. This coloring task, whose sequential version is known to be NP-complete, belongs to a class of **Constraint Satisfaction Problems (CSPs)** that present interesting challenges to distributed computation, particularly in the framework of connectionist architectures. We call the distributed versions of such problems **Network Consistency Problems (NCPs)**. We consider what types of distributed models admit a self-stabilizing algorithm (namely, one that converges to a solution from any initial state of the network and the algorithm), and present such algorithms when possible.

*This work was partially supported by the National Science Foundation, Grant #IRI-8821444 and by the Air Force Office of Scientific Research, Grant #AFOSR-90-0136.

Constraints are useful in programming languages, simulation packages and general knowledge representation systems, and the prospects of solving problems by connectionist networks promise the combined advantages of parallelism, simplicity of design, and error correction capabilities.

Indeed, many interesting problems attacked by researchers in neural networks are combinatorial and many involve constraint satisfaction [1, 6]. In fact, any discrete state connectionist network can be viewed as a type of constraint network, with each stable pattern of states representing a consistent solution. However, current connectionist approaches to CSPs lack theoretical guarantees of convergence (to a solution satisfying all constraints), and the terms on which such convergence can be guaranteed (if at all) have not been systematically explored till now. Other related attempts for solving CSPs distributedly are based on a general constraint propagation [14], thus do not guarantee convergence to a consistent solution.

In this paper we show that widely used connectionist-type architectures in which all nodes run identical procedures cannot admit algorithms that guarantee convergence to a consistent solution, even if such a solution exists (Section 2). We then identify a distributed model that is close in spirit to the connectionist paradigm, for which such guarantees can be established (Section 3). Within this model, we characterize and provide a self-stabilizing algorithm for a restricted subclass of networks that can be solved uniformly (Section 4). Preliminary versions of some of these results first appeared in [3, 4].

2 Model and Definitions

2.1 CSP definition

A network of binary constraints involves a set of n variables $X_1, \ldots, X_n$, each represented by its domain values, $D_1, \ldots, D_n$, and a set of constraints. A **binary constraint** R_{ij} between two variables X_i and X_j is a subset of the Cartesian product $D_i \times D_j$ that specifies which values of the variables are compatible with each other. A **solution** is an assignment of values to all the variables which satisfies all the constraints, and the constraint satisfaction problem associated with such a network is to find one or all solutions. Formally, the solution set for a CSP is defined by: $\rho = \{\bar{x} = x_1, \ldots, x_n | \forall i, j (x_i, x_j) \in R_{ij}\}$.

A binary CSP can be associated with a **constraint graph** in which nodes represent variables and links (edges) connect pairs of variables which are explicitly constrained. (General constraint satisfaction problems may involve constraints of any arity, but since network communication is only pairwise we focus on this subclass of problems.) Figure 1a presents a CSP constraint graph, where each node represents a variable having values $\{a, b, c\}$, and each link is associated with a strict lexicographic order ($X_i \prec X_j$ in the lexicographic order iff $i < j$). The domains are explicitly indicated on the nodes X_3 and X_4 and the constraints are explicitly listed near the link between them, where a pair (m, n) corresponds to possible values for X_3 and X_4, respectively. This means that possible solutions can have $(X_3 = a \wedge X_4 = b)$, $(X_3 = a \wedge X_4 = c)$ or $(X_3 = b \wedge X_4 = c)$.

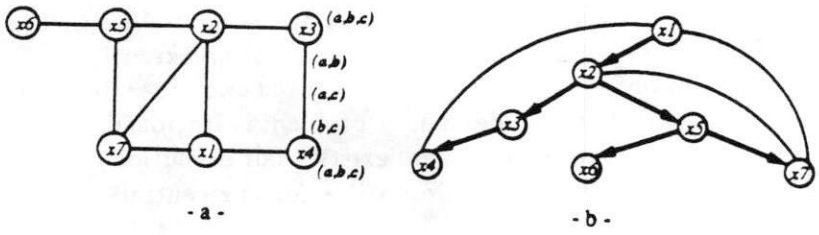

Figure 1: CSP constraint graph and its DFS spanning tree

2.2 The communication model

Our general communication model is known as the **"shared memory multi-reader single-writer"** model. A distributed network consists of n nodes that communicate through shared communication registers. The network can be viewed as a **communication graph** where nodes represent processors and links correspond to communication registers. We assume that the communication and the constraint graphs are identical, and thus two nodes communicate iff they are constrained. A **general** communication register (denoted $state_i$) is written into only by node i, but may be read by all of i's neighbors. A **private** communication register (denoted $r_{i,j}$) is written into by node i and may be read only by one of i's neighbors j. A communication register may have several fields, but is regarded as one unit. A private communication register can be regarded as a field of the general register, accessed only by the appropriate neighbor.

A node can be modeled as a finite state-machine whose state is controlled by a **transition function** that is dependent on its current state and the states of its neighbors. In other words, an activated node performs an **atomic step** consisting of reading the states of all its neighbors (if necessary), deciding whether to change its state and then moving to a new state[1]. A state of the processor encodes the values of its communication registers and its internal variables. A **configuration** c of the system is the state vector of all nodes.

Let c_1, c_2 be two configurations. We write $c_1 \rightarrow c_2$ if c_2 is a configuration which is reached from configuration c_1 by some subset of processors simultaneously executing a single atomic step. An **execution** of the system is an infinite sequence of configurations $E = c_0, c_1 \ldots$ such that for every i, $c_i \rightarrow c_{i+1}$. The **initial configuration** is denoted c_0. An execution is considered **fair** if every node participates in it infinitely often.

We present the transition functions as programs. Assuming that the "program counter" is one of the local variables encoded by the state, an execution of the program step by step is equivalent to a sequence of state transitions. The collection of all transition functions is called a **protocol**. The processors are anonymous, i.e., have no identities (we use the terms node i and processor P_i interchangeably and as a writing convenience only).

The execution of the system can be managed either by **a central demon** (scheduler) defined in [9, 10] or by **a distributed demon** defined in [2, 10]. The distributed demon

[1] In fact, a finer degree of atomicity, requiring only a **test-and-set** operation, is sufficient, but is not used here in order to simplify the arguments.

activates a subset of the system's nodes at each step, while the central demon activates only one node at a time. All activated nodes execute a single atomic step simultaneously. Thus the scheduler affects the possible executions. The central demon can be viewed as a simplified version of the distributed one, since its executions are included in the executions of the distributed demon. A problem is **impossible** for a demon if for every possible program using that demon, there exists a fair execution that does not find a solution to the problem. Therefore, what is impossible for the central demon is impossible also for the distributed one.

When a central demon is assumed, an interleaving of single operations is sufficient for the analysis of the protocol. Nevertheless, on the implementation level, truly independent nodes can execute in parallel since they cannot affect each other. Only neighboring nodes in the communication graph cannot execute at the same atomic step when a central demon is assumed.

2.3 Self-stabilization

A self-stabilizing protocol [9] is one with a particular convergence property. The system configurations are partitioned into two classes — legal, denoted by L, and illegal. The protocol is self-stabilizing if in any infinite fair execution, starting from any initial configuration (and with any input values) and given "enough time", the system eventually reaches a legal configuration and all subsequently computed configurations are legal. Thus a self-stabilizing protocol converges from any point in its configuration space to a stable, legal region. Note that the lack of any assumption about the initial configuration also means that a node can start the execution of the protocol at any point, not necessarily from the beginning.

The legality of a configuration depends on the aim of the protocol. In our case, we wish to design a protocol for solving the network consistency problem. Thus, the set of legal configurations are those having a consistent assignment of values to all the nodes in the network, if such an assignment exists, and any set otherwise. This definition allows the system to oscillate among various solutions, if more than one consistent assignment is possible. However, the protocols that are presented in this paper converge to one of the possible solutions.

An inherent limitation of the self-stabilizing model is that a global view of the entire network is needed in order to detect whether a legal configuration has been reached. A node is not able to identify (via its state and the states of its neighbors) whether the convergence of the system has been completed. Therefore, self-stabilizing protocols are assumed to run forever, repeatedly checking for legality, even though eventual stabilization of the system is guaranteed and at some point the execution could be externally terminated.

Since internal detection of global convergence is impossible, composition of two self-stabilizing subprotocols cannot be implemented by executing one of them and after its convergence executing the other. Nevertheless, composition of self-stabilizing protocols is easily implemented by an interleaved execution since the correctness proof of the resulting protocol is implied by the correctness of its self-stabilizing components. If subprotocol $\mathcal{A}$ depends on the results of subprotocol $\mathcal{B}$, the execution of $\mathcal{A}$ is likely to be meaningless before the convergence of $\mathcal{B}$. However, $\mathcal{A}$'s convergence is eventually guaranteed, since $\mathcal{B}$ eventually

converges, and following that point the assumption used in the correctness proof of $\mathcal{A}$ is true, and $\mathcal{A}$ will converge. Similar observations have been made independently in [10].

2.4 The limits of uniform self-stabilization

A protocol is **uniform** if all the nodes are logically equivalent and identically programmed (i.e., have identical transition functions). Following an observation made by Dijkstra [9] regarding the computational limits of a uniform model for performing the mutual exclusion task, we show that the network consistency problem cannot be solved using a uniform protocol. This is accomplished by presenting a specific constraint network and proving that its convergence cannot be guaranteed using any uniform protocol.

Consider the task of numbering a ring of processors in a cyclic ascending order — we call this CSP the **"ring ordering problem"**. The constraint graph of the problem is a ring of nodes, each with the domain $\{0, 1, \ldots, n-1\}$. Every link has the set of constraints $\{(i, (i+1) \bmod n) | \ 0 \leq i \leq n-1\}$ i.e., the left node is smaller by one than the right. A solution to this problem is a cyclic permutation of the numbers $0, \ldots, n-1$, which means that there are n possible solutions, and in all of them different nodes are assigned different values.

Theorem 1: No uniform, self-stabilizing protocol can solve the ring ordering problem, under a central demon.

Proof: In order to obtain a contradiction, assume that there exists a uniform self-stabilizing protocol for solving the problem. In particular, it would solve the ring ordering problem for a ring having a composite number of nodes, $n = r \cdot q$ $(r, q > 1)$. Since convergence to a solution is guaranteed from any initial configuration, in particular, the protocol converges when initially all nodes are in identical states. We construct a fair execution of such a protocol for which the network never converges to a consistent solution, contradicting the self stabilization property of the protocol. Assume the following execution:

$$
\begin{array}{llllll}
P_0, & P_q, & P_{2q}, & \ldots, & P_{(r-1)q}, \\
P_1, & P_{q+1}, & P_{2q+1}, & \ldots, & P_{(r-1)q+1}, \\
\vdots & & & & \\
P_{q-1}, & P_{2q-1}, & P_{3q-1}, & \ldots, & P_{rq-1}, \\
P_0, & \ldots & & & \\
\vdots & & & &
\end{array}
$$

Note that nodes $P_0, P_q, P_{2q}, \ldots, P_{(r-1)q}$ move to identical states, after their first activation, because their inputs, initial states and transition functions are identical, and when each one of them is activated its neighbors are in identical states too. The same holds for any sequential activation of processors $\{P_{iq+j} | \ 0 \leq i < r, \ 0 \leq j < q\}$. Thus, cycling through the above schedule assures that P_0 and P_q, for instance, move to identical states over and over again, an infinite number of times. Since a consistent solution requires their states to be different, the network will never reach a consistent solution, thus yielding a contradiction. Figure

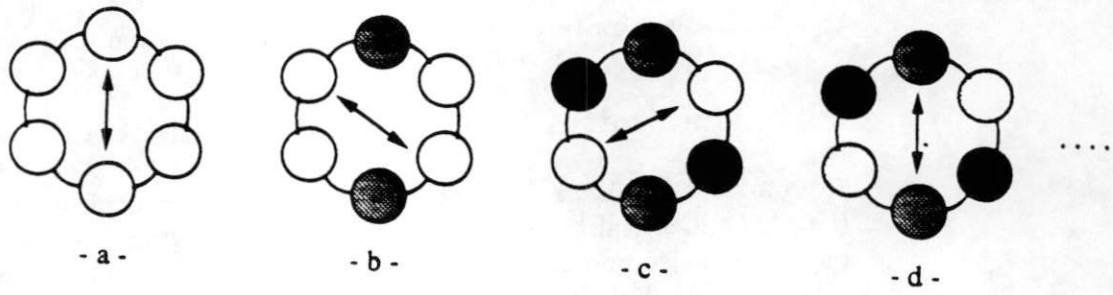

Figure 2: The ring ordering problem ($n = 6$)

2 demonstrates such a counterexample execution for a ring with six nodes. The indicated nodes are scheduled in each configuration. Different colors refer to different states. □

Theorem 1 is proven above for a centralized demon, but it holds also for a distributed demon, which can produce the same schedule. Theorem 1 implies that it is generally impossible to guarantee convergence to a consistent solution using a uniform protocol. It also implies that such convergence cannot be guaranteed for a class of sequential algorithms using so called "repair" methods, such as in [17]. It does not, however, exclude the possibility of uniform protocols for restricted activation policies or for particular net architectures (not including a ring).

We can also show that, when using a distributed demon, convergence (to a solution) cannot be guaranteed even for **tree networks**. Consider, for instance, the coloring problem in a tree-network constructed from two connected nodes, each having the domain {BLACK, WHITE}. Since the two nodes are topologically identical, if they start from identical initial states and both of them are activated simultaneously, they can never be assigned different values. Consequently, the network does not converge to a legal solution, although one exists. This counter-example can be extended to a large class of trees, where there is no possible way to distinguish between two internal nodes. However, we will show, (Section 4) that for a central demon, a uniform self-stabilizing tree network consistency protocol does exist.

Having proven that the network consistency problem cannot be solved using a uniform protocol, even with a central demon, we switch to a slightly more relaxed model of an "**almost uniform**" protocol, in which all nodes but one are identical. We denote the special node as node 0 (or P_0).

3 Consistency-Generation Protocol

This section presents an almost uniform, self-stabilizing network consistency (NC) protocol. The completeness of this protocol (i.e., the guarantee to converge to a solution) if one exists, stems from the completeness of the sequential constraint satisfaction algorithm it simulates. We briefly review some basic sequential techniques for constraint satisfaction.

3.1 Sequential aspects of constraint satisfaction

The most common algorithm for solving a CSP is **backtracking**. In its standard version, the algorithm traverses the variables in a predetermined order, provisionally assigning consistent values to a subsequence $(X_1, \ldots, X_i)$ of variables and attempting to append to it a new instantiation of X_{i+1} such that the whole set is consistent (**"forward"** phase). If no consistent assignment can be found for the next variable X_{i+1}, a deadend situation occurs; the algorithm "backtracks" to the most recent variable (**"backward"** phase), changes its assignment and continues from there.

One useful improvement of backtracking, called **backjumping** [7] consults the topology of the constraint graph to guide its backward phase. Specifically, instead of going back to the most recent variable instantiated, it *jumps back* several levels to the first variable connected to the deadend variable. If the variable to which the algorithm retreats has no more values, it backs up further, to the most recent variable among those connected either to the original variable or to the new deadend variable, and so on.

It turns out that when using a depth-first search (DFS) on the constraint graph (to generate a DFS spanning tree) and then conducting backjumping in an inorder traversal of the DFS tree [11], the jump-back destination of variable X is the parent of X in the DFS spanning tree.

The nice property of a DFS spanning tree that allows a parallel implementation is that any arc of the graph which is not in the tree connects a node to one of its tree ancestors (i.e., to a node residing along the path leading to it from the root). Consequently, the DFS spanning tree represents a useful decomposition of the graph: if a variable X and all its ancestors in the tree are removed from the graph, the remaining subtrees rooted at children of X will be disconnected. Figure 1b presents a DFS spanning tree of the constraint graph presented in Figure 1a. Note that if X_2 and its ancestor X_1 are removed from the graph, the network becomes two disconnected trees rooted at X_3 and X_5. This translates to a problem decomposition strategy: if all ancestors of variable X are instantiated, then the solutions of all its subtrees are completely independent and can be performed in parallel [12].

3.2 General protocol description

Our network consistency protocol is based on a distributed version of the sequential backjumping algorithm, implemented on a variable ordering generated by a depth-first traversal of the constraint graph.

The NC protocol is logically composed of two self-stabilizing subprotocols that can be executed interleaved, as explained in Section 2.3:

1. DFS spanning tree generation
2. value assignment (using the graph traversal mechanism)

The second subprotocol assumes the existence of a DFS spanning tree in the network. However, the implementations of these subprotocols are unrelated to each other and, thus, can be independently replaced by any other implementation.

Until the first subprotocol establishes a DFS spanning tree, the second subprotocol will execute, but in all likelihood will not lead to a proper assignment of values. However, we prove that the DFS spanning tree generation subprotocol is self-stabilizing, and thus generation of a DFS spanning tree is eventually guaranteed. The convergence of the second subprotocol is also guaranteed starting from any configuration, assuming the existence of a DFS spanning tree. Therefore, it is guaranteed to converge properly after the DFS spanning tree generation has been completed.

The basic idea of the protocol is to decompose the network (problem) logically into several independent subnetworks (subproblems), according to the DFS spanning tree structure, and to instantiate these subnetworks (solve the subproblems) in parallel. A proper control over value instantiation is guaranteed by the graph traversal mechanism presented in Section 3.4.

Below are some of the notations we use through the rest of the paper. When the DFS spanning tree generation subprotocol stabilizes, each internal node, i, has one adjacent node, $parent(i)$, designated as its **parent** in the tree, and a set of **child** nodes denoted $children(i)$. Figure 3 indicates the environment of an internal node (3a), the root (3b), and a leaf (3c). The link leading from $parent(i)$ to i is called i's **inlink** while the links connecting i to its children are called i's **outlinks**. The set of i's neighboring nodes in the graph that are also its **ancestors** in the DFS spanning tree (i.e., reside along the path from the root to i) are called i's **predecessors** and denoted $predecessors(i)$.

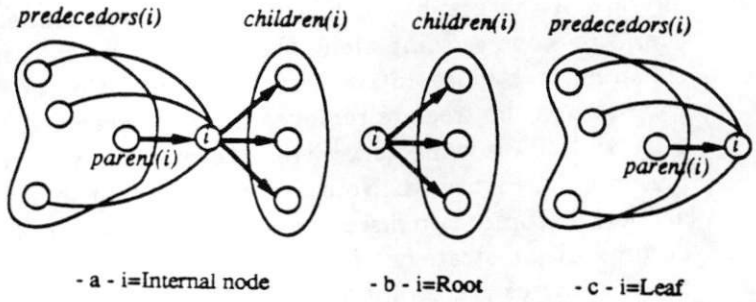

Figure 3: The neighborhood set of a node

3.3 Self-stabilizing DFS spanning tree generation protocol

This section presents an almost uniform, self-stabilizing protocol for generating a DFS spanning tree. The protocol, its correctness proof, and some generalizations appear in [5]. However, for the sake of completeness we present the protocol and a sketch of the proof in this paper. This subprotocol is the source of non-uniformity for the whole NC protocol. The root of the generated tree will be the distinguished node 0 (P_0).

3.3.1 Minimally labeled tree

For the sake of tree generation, we assume that every node i has an enumerating function α_i, which enumerates all i's adjacent edges in some order. We use the following definitions

that apply to every node in the graph:

edge number – the value $\alpha_i(j)$ given by node i to the edge that leads to its neighbor j. Every edge has a number when viewed from one end, and perhaps a different number when viewed from the other end.

label – a sequence of edge numbers.

valid label – a sequence of edge numbers starting with $\perp$ that describes a simple (loopless) path from the root to the node, represented by the edge numbers in the same order as they appear along the path.

minimal label – the smallest valid label of a node with respect to the complete lexicographical order "$\prec$", defined over the set of labels, where $\perp$ is the minimal character. We denote the minimal label of node i as ℓ_i. A node that has its minimal label is said to be **minimally labeled.**

minimal path – the path from the root to a node that is represented by the minimal label.

Lemma 1: The set of the minimal labels of all the nodes in the graph induces a spanning tree called the **minimally labeled tree.**

Proof: Every minimal label induces (represents) a path. In order to prove the lemma, we show that any combination of the minimal paths does not create loops (when directions are ignored). Obviously, there are no loops in a single minimal path, since otherwise its representing label wouldn't be minimal. Moreover, there are no two minimal paths to node i and to node j with different subpaths leading from the root to the same node k. This would mean that the minimal labels ℓ_i and ℓ_j have different prefixes leading to k. At least one of these prefixes is greater than ℓ_k. Hence replacing it with ℓ_k would decrease the label, contradicting its minimality. Thus in the set of all the minimal paths there are no two different subpaths leading from the root to the same node, and therefore no loop is created by combining minimal paths. Since the minimal paths connect all the nodes to the root, their combination produces a connected, loopless subgraph, which is a spanning tree. □

Theorem 2: The minimally labeled tree is a DFS spanning tree.

Proof: We prove the theorem by showing that for every two neighboring nodes in the graph one is an ancestor (predecessor in our terms) of the other in the minimally labeled tree. This is one definition of a DFS spanning tree. For full details see [5]. □

Next we present a protocol that generates a minimally labeled tree in the graph.

3.3.2 DFS spanning tree protocol

Generally the DFS spanning tree generating protocol works as follows: During the execution of the protocol the special node 0, which plays the role of the root, assigns itself the label $\perp$ and suggests labels to its neighbors. The label suggested to j by its neighbor i is constructed by concatenating $\alpha_i(j)$ to i's own label ($\perp$ in the case of the root). Every non-root node chooses the smallest label suggested to it to be its label and the suggesting neighbor to be its parent, and suggests labels to its neighbors in a similar way.

The communication register between i and j contains the following fields used by the DFS spanning tree generation protocol:

$r_{ij}.mark$ – contains the label that is "suggested" to j by i.

$r_{ij}.par$ – a boolean field that is set to TRUE iff i chose j as its parent.

The edges of the tree can be regarded as directed from the children towards the parent, and the children set of node i is computed by selecting the neighbors whose $r_{ji}.par$ field is true. Additionally every node has two internal variables:

L_i – contains the label of i.

$parent_i$ – contains the number of the edge according to α_i that leads to the neighbor selected as i's parent.

We prove that in this protocol the labels of all the nodes (namely, the values of the L_i variables) eventually converge to the minimal labels (ℓ_i-s), and the tree that is constructed from the edges that are indicated by the *parent* variables is the minimally labeled tree.

We assume that all the variables have a fixed length (number of bits). Assigning a too large value to a variable causes an overflow that results in losing the most significant part of the value. This is an abstraction of the common overflow bit mechanism, which is used to simplify the presentation and the correctness proof of the protocol. The notation $\xleftarrow{|n|}$ refers to the assignments where the above assumption is essential to the correctness of the protocol, to express that no more than n least significant bits are assigned. To guarantee the correctness of our DFS spanning tree protocol, we assume that $n \geq v \log k$, where v is the number of the nodes in the graph and k is the maximal degree. This is the number of bits that are needed to represent the largest minimal label that is possible in the graph. Figure 4 presents the protocol.

Theorem 3: After a finite stabilization period, all the nodes are minimally labeled in their L variables, the *parent* variable of every node points to its parent in the minimally labeled tree and there are no further changes in the values of these variables.

Proof: After the first execution of the algorithm by a node, its *parent* variable is always equal to the last character of its label. Moreover, eventually there is an equivalence between the set of the labels in the graph and the tree induced by the *parent* variables. Thus, it is

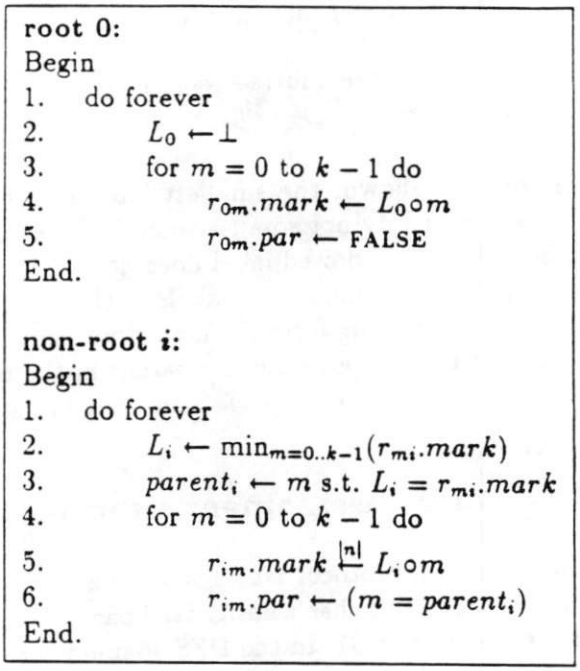

Figure 4: DFS spanning tree protocol

sufficient to prove the convergence of the labels (L values) to the minimal labels. We prove this convergence by induction on the depth of the node in the minimally labeled tree.
Base: $d = 0$. The root assigns the value $\perp$ to its L variable, which is its minimal label by definition.
Step: Assume that the labels of all nodes whose depth in the minimally labeled tree is smaller than d have converged to the minimal labels and that these nodes have written their suggestions. Consider a node i whose depth in the minimally labeled tree is d. We prove that after finite time the following assertion is satisfied: the labels of all the nodes in the subtree rooted at i in the minimally labeled tree is $\succeq \ell_i$. Let j (possibly $j \equiv i$) be the node with the smallest label among the nodes in the subtree rooted at i, after the nodes with depth not greater than $d-1$ have stabilized and after all the nodes have updated their values at least once. The last assumption assures that from this point on every node executes the protocol from the beginning, and, as we are about to show, the smallest label value in the subgraph will not decrease.

Since the minimally labeled tree is a DFS spanning tree, the edges connect the nodes in the subtree rooted at i (and j in particular) either to i's ancestors, whose labels have already converged (using the induction hypothesis), or to nodes that are in the same subtree. Consider the next time L_j and $parent_j$ are updated. Let k be the neighbor that j chooses to be its parent at this stage:

1. If k is i's ancestor in the minimally labeled tree, then L_k has already converged to ℓ_i, and thus by the definition of ℓ_j, $\alpha_k(j)$ is greater (or equal if $j \equiv i$) than the number

given by α_k to the first edge on the path from k to i. Therefore $L_j \stackrel{|n|}{\leftarrow} L_k \circ \alpha_k(j) \succeq \ell_i$. The label of j increases, unless it is already not smaller than ℓ_i.

2. If k is in the subtree rooted at i then $L_j \stackrel{|n|}{\leftarrow} L_k \circ \alpha_k(j)$, which means that j's label increases.

As we have shown, the smallest label in the subtree rooted at i increases, unless all the labels are already not smaller than ℓ_i. After a finite number of incrementations an overflow appears. An overflowed label does not begin with $\perp$ and thus it is $\succ \ell_i$. Therefore, after a finite period of time, all the labels in the subtree rooted at i are not smaller than ℓ_i and stay this way. Beginning from the next time i applies its algorithm, all the suggestions it gets are $\succeq \ell_i$, and the suggestion of its parent in the minimally labeled tree is ℓ_i (since the parent's label has already converged). Thus, beginning from the next time L_i is updated, $L_i = \ell_i$. □

3.4 Value assignment subprotocol

The second subprotocol assumes the existence of a DFS spanning tree in the network, namely, each non-root node has a designated parent, children, and predecessors among its neighboring nodes (see Figure 3). In the DFS spanning tree generation subprotocol presented in Section 3.3, each node has a minimal label and a designated parent. Using this information, each node can compute its children set, $children(i)$, by selecting the neighbors whose $r_{ji}.par$ field is true, and its predecessors set, $predecessors(i)$, by selecting the neighbors whose minimal label ($r_{ji}.mark$ without the last character) is a prefix of its own. This means that we can view the directed tree as directed both towards the leaves and towards the root.

The value assignment subprotocol presents a graph traversal mechanism that passes control to the nodes in the order of the value assignment of the variables (in DFS order), without losing the parallelism gained by the DFS structured network. Section 3.4.1 presents the basic idea of **privilege passing** that implements the graph traversal mechanism, while Section 3.4.2 presents the value assignment strategy that guarantees convergence to a solution.

Each node i (representing variable X_i) has a list of possible values, denoted as $Domain_i$, and a pairwise relation R_{ij} with each neighbor j. The domain and the constraints may be viewed as a part of the system or as inputs that are always valid (though they can be changed during the execution, forcing the network to readjust itself to the changes).

The state register of each node contains the following fields:

value$_i$ – a field to which it assigns one of its domain values or the symbol '$\star$' (to denote a deadend).

mode$_i$ – a field indicating the node's "belief" regarding the status of the network. A node's mode is ON if the value assignment of itself or one of its ancestors was changed since the last time it was in a forward phase (to be explained in Section 3.4.2), or otherwise it is OFF . The modes of all nodes also give an indication of whether they have reached a consistent state (all in an OFF mode).

parent_tag and ***children_tag*** – two boolean fields that are used for the graph traversal (Section 3.4.1).

Additionally, each node has a sequence set of domain values that is implemented as an ordered list and is controlled by a local domain ***pointer*** (to be explained later), and a local ***direction*** field indicating whether the algorithm is in its forward or backward phase.

3.4.1 Graph traversal using privileges

The graph traversal is handled by a self-stabilizing **privilege passing mechanism**, according to which a node obtains the privilege to act, granted to it either by its parent or by its children. A node is allowed to change its state only if it is privileged.

Our privilege passing mechanism is an extension of a mutual exclusion protocol for two nodes called **balance-unbalance** [9, 10]. Once a DFS spanning tree is established, this scheme is implemented by having every state register contain two fields: *parent_tag*, referring to its inlink and *children_tag*, referring to all its outlinks. A link is **balanced**, if the *children_tag* and the *parent_tag* on its endpoints have the same value, and the link is **unbalanced** otherwise. A node becomes privileged if its inlink is unbalanced and **all** its outlinks are balanced. In other words, the following two conditions must be satisfied for a node i to be privileged:

1. $parent\_tag_i \neq children\_tag_{parent(i)}$ (the inlink is unbalanced)
2. $\forall k \in children(i) \; : \; children\_tag_i = parent\_tag_k$ (the outlinks are balanced)

By definition, we consider the inlink of the root to be unbalanced and the outlinks of the leaves to be balanced.

A node applies the value assignment subprotocol (described in Section 3.4.2) only when it is privileged, otherwise it leaves its state unchanged. As part of the execution of the subprotocol, the node passes the privilege. The privilege can be passed backwards to the parent by balancing the inlink or forward to the children by unbalancing the outlinks (i.e., by changing the value of *parent_tag* or *children_tag*, respectively).

We use the following notations to define the set of configurations that are legally controlled relatively to the graph traversal:

- Denote a **chain** to be a maximal sequence of unbalanced links, $e_1, e_2 \ldots, e_n$, s.t. :
  1. the inlink of the node whose outlink is e_1, is balanced, unless the node is the root.
  2. every adjacent pair of links e_i, e_{i+1} $(1 \leq i < n)$ is an inlink and an outlink, respectively, of a common node.
  3. all the outlinks of the node whose inlink is e_n, are balanced.
- The chain begins at the node with e_1 as one of its outlinks denoted as the **chain head**, and ends at the node with the inlink e_n, denoted as the **chain tail**.

- Denote a **branch** to be a path from the root to a leaf.
- A branch **contains** a chain (or a chain is **on** the branch) if all the links of the chain are in the branch.
- A configuration is **legally controlled** if it does not contain any non-root chain heads, namely, every branch of the tree contains no more than one chain and its chain head is the root.

Figure 5 shows a legally controlled configuration. The DFS spanning tree edges are directed, and the values (+ or −) of the *parent_tag* and the *children_tag* of every node are specified above and below the node, respectively. The privileged nodes are black. In a legally controlled configuration a node and its ancestors are not privileged at the same time and therefore cannot reassign their values simultaneously. The privileges travel backwards and forwards along the branches. We prove (Section 3.4.3) that using the graph traversal mechanism, the network eventually converges to a set of legally controlled configurations that are also legal with respect to the network consistency task.

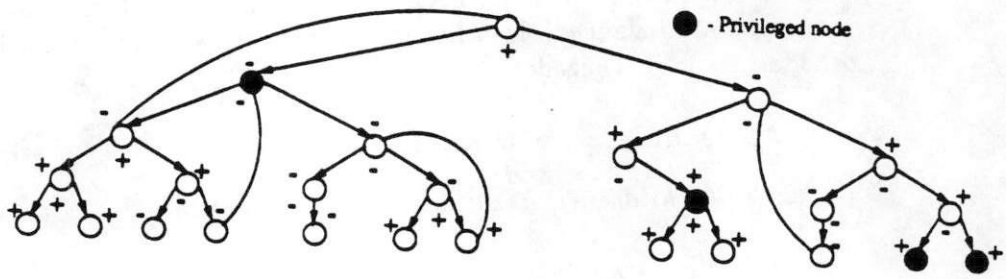

Figure 5: Legally controlled configuration

3.4.2 Value assignment

The value assignment has forward and backward phases, corresponding to the two phases of the sequential backtracking algorithm. During the forward phase, nodes in different subtrees assign themselves (in parallel) values consistent with their predecessors or verify the consistency of their assigned values. When a node senses a deadend, that is it has no consistent value to assign, it assigns its *value* field a '$\star$' and initiates a backward phase. Since the root has no ancestors, it does not check consistency and is never deadended. It only assigns a new value at the end of a backward phase, when needed, and then initiates a new forward phase.

When the network is consistent (all the nodes are in an OFF *mode*), the forward and backward phases continue, where the forward phase is used to verify the consistency of the network, while the backward phase just returns the privilege to the root to start a new forward wave. Once consistency is violated, the node sensing the violation relative to its

predecessors, moves to an ON mode and initiates a new value assignment. A more elaborate description follows.

An internal node can be in one of three situations:

- **Node i is activated by its parent which is in an ON mode** (this is the forward phase of value assignments). In that case some change of value in one of its predecessors might have occurred. It, therefore, finds the first value in its domain that is consistent with all its predecessors, puts itself in an ON mode and passes the privilege to its children. If no consistent value exists, it assigns itself the '$\star$' value (a deadend) and passes the privilege to its parent (initiating a backward phase).

- **Node i is activated by its parent which is in an OFF mode.** (this is the forward phase of consistency verification). In that case it verifies the consistency of its current value with its predecessors. If it is consistent it stays in (or moves to) an OFF mode and passes the privilege to its children. If not, it tries to find the next value in its domain that is consistent with all its predecessors, and acts like in the previous case. A leaf, having no children, is always activated by its parent and always passes the privilege back to its parent (initiating a backward phase).

- **Node i is activated by its children** (backward phase). If one of the children has a '$\star$' value, i selects the next value in its domain that is consistent with all its predecessors, and passes the privilege back to its children. If no consistent value is available, it assigns itself a '$\star$' and passes the privilege to its parent. If all children have a consistent value, i passes the privilege to its parent.

Due to the privilege passing mechanism, when a parent sees one of its children in a deadend it still has to wait until **all** of them have given it the privilege. This is done to guarantee that all subtrees have a consistent view regarding their predecessors' values.

Once it has become privileged, a node cannot tell where the privilege came from (i.e., from its parent or from its children). Thus, a node uses its *direction* field to indicate the source of its privilege. Since during the legally-controlled period no more than one node is privileged on every branch, the privileges travel along the branches backwards and forwards. The *direction* field of each node indicates the direction of the next expected wave. When passing the privilege to its children, the node assigns its *direction* field the BACKWARD value, expecting to get the privilege back during the next backward wave, while when passing the privilege to its parent it assigns the FORWARD value, preparing itself for the next forward wave. Thus, upon receiving the privilege again, it is able to recognize the direction it came from: if *direction* = BACKWARD, the privilege was recently passed towards the leaves and therefore it can come only from its children; if *direction* = FORWARD, the privilege was recently passed towards the root and therefore it can come only from its parent. The value of the *direction* field can be improper upon the initialization of the system. However, after the first time a node passes the privilege its *direction* field remains properly updated. Figure 6 presents the privilege passing procedures for node i.

The algorithms performed by a non-root node ($i \neq 0$) and the root once they become privileged and after reading the neighbors' states are presented in Figures 7 and 8.

```
procedure pass-privilege-to-parent
Begin
1.  direction_i ← FORWARD                          { prepare for the next wave }
2.  parent_tag_i ← children_tag_parent(i)          { balance inlink }
End.

procedure pass-privilege-to-children
Begin
1.  direction_i ← BACKWARD                         { prepare for the next wave }
2.  children_tag_i ← ¬parent_tag_k∈children(i)     { unbalance outlinks }
End.
```

Figure 6: Privilege passing procedures

```
root 0:
Begin
1.  if ¬consistent(value_0, children(0)) then
2.      mode ← ON
3.      value ← next-value
4.  else                              { all children are consistent }
5.      mode ← OFF
6.  pass-privilege-to-children
End.

non-root i:
Begin
1.  if direction = FORWARD then       { forward phase }
2.      if mode_parent(i) = ON then   { a change in a value assignment occurred }
3.          pointer ← 0               { reset domain pointer }
4.      else                          { parent's mode is OFF }
5.          if consistent(value_i, predecesors(i)) then
6.              mode ← OFF
7.  if (pointer = 0) ∨ ¬consistent(value_i, predecesors(i)) ∨
                ∨ (direction = BACKWARD ∧ ¬consistent(value_i, children(i))) then
8.      value ← compute-next-consistent-value
                                      { privilege passing }
9.  if leaf(i) ∨ (value = ⋆) ∨ (direction = BACKWARD ∧ consistent(value_i, children(i)))
10.     then pass-privilege-to-parent
11.     else pass-privilege-to-children
End.
```

Figure 7: Value assignment subprotocols for root and non-root nodes

The procedure **compute-next-consistent-value** (Figure 8) tests each value located after the domain pointer for consistency. More precisely, the domain value is checked against each of $predecessor(i)$'s values, and the next domain value consistent with the predecessors is returned. The pointer's location is readjusted accordingly (i.e., to the found value) and the mode of the node is set to ON . If no consistent value is found, the value returned is '$\star$' and the pointer is reset to the beginning of the domain. The predicate **consistent**(val, set_of_nodes) is TRUE if the value of val is consistent with the *value* fields of set_of_nodes and none of them is deadended (has the value '$\star$').

```
procedure compute-next-consistent-value
Begin
1.   mode_i ← ON
2.   while pointer ≤ endof Domain_i do
3.        pointer ← pointer + 1
4.        if consistent(Domain_i[pointer], predecessors(i)) then
5.             return Domain_i[pointer]     { a consistent value was found }
6.   pointer ← 0
7.   return ⋆                { no consistent value exists }
End.
```

Figure 8: Consistency procedure

The algorithm performed by the root, P_0, when it is privileged, is slightly different and in a way simpler. The root does not check consistency. All it does is assign a new value at the end of each backward phase, when needed, and then initiate a new forward phase. The procedure **next-value** increments the domain pointer's location and returns the value indicated by the domain pointer. If the end of the domain list is reached, the pointer is reset to the first (smallest) value.

The value assignment subprotocol can be regarded as uniform since each node may have both the root's protocol and the non-root's protocol and decide between them based on the role assigned to it by the DFS spanning tree protocol.

3.4.3 Proof of self-stabilization

To prove the correctness of our NC protocol, we first prove that the graph traversal is self-stabilizing, namely, that the system eventually reaches a legally controlled configuration (even if the values in the nodes are not yet consistent), and from that point it remains legally controlled. Assuming the system is legally controlled, we show that if a legal assignment exists, it is eventually reached and thereafter remains unchanged. Thus the system reaches a legal set of configurations and stays there — and therefore is self-stabilizing.

Before presenting the proof, note that a non-root node is privileged when its inlink is unbalanced (thus it is on a chain) and all its outlinks are balanced. In other words, a non-root

node is privileged iff it is a chain tail. The root is privileged iff it is not a chain head. Also note that passing the privilege by a node affects only the chains on the branches containing that node, because it has no interaction with other branches.

In order to prove the self-stabilization of the privilege passing mechanism, we first prove some of its properties.

Lemma 2: In every infinite fair execution, every non-root node that is on a chain eventually passes the privilege to its parent.

Proof: We prove the lemma by induction on the node's height, h (i.e., its distance from the nearest leaf), and on the possible value assignments from the domain of the node.

Base: $h = 0$. The node is a leaf and, therefore, when activated, can pass the privilege only to its parent.
Step: Assume node i, whose height is $h > 0$, is on a chain. Node i eventually becomes privileged, because if any of i's outlinks are unbalanced, then the corresponding children are on chains and the induction hypothesis holds for them, namely they eventually pass the privileges to i. Note that a node that passes its privilege to the parent (to i in our case) does not become privileged again, unless its parent had become privileged first and passed the privilege to its children, since outlinks are unbalanced by a privileged node only.

If, when becoming privileged, i passes the privilege to its parent, the claim is proven. Otherwise, whenever i passes the privilege to its children the same argument holds, so i eventually becomes privileged again. Moreover, i's domain pointer is advanced every time it passes the privilege to its children. Therefore, after a finite number of such passings, bounded by the size of $domain_i$, the domain pointer reaches a '$\star$' and then, following the code in Figures 7 and 8, i passes the privilege to its parent. □

Theorem 4: The graph traversal mechanism is self-stabilizing with respect to the set of legally controlled configurations. Namely, it satisfies the following assertions:

1. *Reachability* – Starting from any initial configuration, the system eventually reaches a legally controlled configuration.
2. *Closure* – If c is a legally controlled configuration and $c \rightarrow c'$ then c' is also a legally controlled configuration.

Proof: We prove the theorem by showing that all the non-root chain heads in the network eventually disappear. Note that passing a privilege by the root makes the root a chain head, but does not increase the number of non-root chain heads. Passing a privilege by any non-root node does not create any chain head. When a non-root node passes the privilege it is a chain tail (its outlinks are balanced). Thus, if the privilege is passed to the node's children, none of them become a chain head, since their parent is still on the same chains. On the other hand, if the privilege is passed to its parent, the node balances its inlink, which cannot possibly create a new chain head. Thus the number of the non-root chain heads in the network never increases. Moreover, Lemma 2 implies that every non-root node

that is on a chain and particularly any non-root chain head eventually passes the privilege to its parent, and stops being a chain head. Therefore, the number of the non-root chain heads steadily decreases, until no non-root chain heads are left, hence the network is legally controlled. Since no non-root chain heads are ever created, the network remains legally controlled forever. □

The self-stabilization property of the NC protocol is inherited from its subprotocols: DFS spanning tree generation and value assignment. Once the self-stabilization of privilege-passing is established, it assures that the control is a correct, distributed implementation of DFS-based backjumping, which guarantees the convergence of the network to a legal solution, if one exists, and if not, repeatedly checks all the possibilities.

3.4.4 Complexity analysis

A worst case complexity estimate of the value assignment subprotocol, assuming a DFS-tree already exists, can be given by counting the number of state changes until the network becomes legally controlled and the additional number of state changes before a legal configuration is reached.

Let T_m^1 stand for the maximal number of privilege passings in the subnetwork with a DFS spanning subtree of depth m, before its root completes a round of assigning all of its domain values (if necessary) and passing the privilege forward to its children for every assigned value (for a non-root node it is the number of privilege passings in its subtree before the node passes the privilege backwards). Let b be the maximal branching degree in the DFS spanning tree and let k bound the domain sizes. Since every time that the root of the subtree becomes privileged it either tries to assign a new value or passes the privilege backwards, T_m^1 obeys the following recurrence:

$$\begin{aligned} T_m^1 &= k \cdot T_{m-1}^1 \\ T_0^1 &= 1 \end{aligned}$$

Solving this recurrence yields: $T_m^1 = k^m$, which is the number of privilege passings before reaching the legally controlled configuration where only the root is privileged.

The worst-case number of additional state changes towards a total convergence depends on the worst-case time of the sequential backjumping algorithm, and the same time bound applies to both of them. Let T_m^2 stand for the maximal number of value reassignments in the subnetwork with a DFS spanning subtree of depth m, before it reaches a solution, which equals the search space that is explored by the sequential algorithm. Since any assignment of a value to the root node generates b subtrees of depth $m-1$ or less that can be solved independently, T_m^2 obeys the following recurrence:

$$\begin{aligned} T_m^2 &= k \cdot b \cdot T_{m-1}^2 \\ T_0^2 &= k \end{aligned}$$

Solving this recurrence yields: $T_m^2 = b^m k^{m+1}$.

Thus the overall worst time complexity of the value assignment subprotocol is: $T_m = T_m^1 + T_m^2 = O(b^m k^m)$. Note that when the tree is balanced we get that $T_m = O(nk^m)$.

We gain additional speedup by exploiting the parallelism. For instance, if all the nodes are scheduled each time, all the subtrees are resolved in parallel which yields $T'^2_m = k^{m+1}$ and thus $T'_m = O(k^m)$.

In summary, we have shown that our protocol's complexity is exponentially proportional to the depth of the DFS spanning tree, i.e., the system has a better chance for a "quick" convergence when the DFS spanning tree is of a minimal depth. Our bound also improves the one presented in [12].

The average performance of the NC protocol can be further improved by adding to it a uniform self-stabilizing **arc-consistency** subprotocol [16]. A network is said to be **arc consistent** if for every value in each node's domain there is a consistent value in all its neighbors' domains. Arc consistency can be achieved by a repeated execution of a "relaxation procedure", where each node reads its neighbors' domains and eliminates any of its own values for which there is no consistent value in one of its neighbors' domains. This protocol is clearly self-stabilizing, since the domain sizes are finite, and they can only shrink or be left intact by each activation. As a result, after a finite number of steps all the domains remain unchanged.

3.5 Example

We demonstrate the NC protocol on a simple network that has to perform a coloring task. Figure 9 presents the initial configuration of the network and the structure of a node. Each node has a domain of three colors, $\{a, b, c\}$, and the constraint '$\neq$' with each of its neighbors. All the nodes start with the same color, a, and the domain pointers are reset to the beginning of the domain lists. For ease of explanation, in this example we assume that all the nodes execute the NC protocol from the beginning. In order to facilitate understanding of the execution, a field indicating whether the node regards itself as a leaf is added to every node. The parent field of a node is indicated as a direction (arrow) on the appropriate edge. In the initial configuration none of the nodes has a parent. The edge numbers are indicated on the endpoints of every edge, and the \$ sign in the labels plays the role of $\perp$. The maximal length of valid labels is assumed to be four digits, including $\perp$. The dashed arrows in Figure 9 identify the minimally labeled tree of the network, while the dashed arrows in Figures 10 and 11 identify the links that are believed to be in the minimally labeled tree in every configuration.

Figures 9, 10, 11 present a prefix of execution composed of six configurations, demonstrating the convergence of the network. In every step all the nodes are scheduled simultaneously. They first read the states of their neighbors, then perform the DFS spanning tree generation procedure, use its result and the previously read values to decide which nodes are privileged, and perform the value assignment procedure accordingly. Finally they update their states, which moves the network to the next configuration.

The nodes that regard themselves as privileged after performing the DFS spanning tree generation procedure are emphasized in every configuration. Note that since there are no parents in the initial configuration (Figure 9), all the non-root nodes believe (after reading their neighbors' states and executing the DFS spanning tree generation procedure) that they are leaves (Figure 10a). This causes redundant privileged nodes. Also note that the

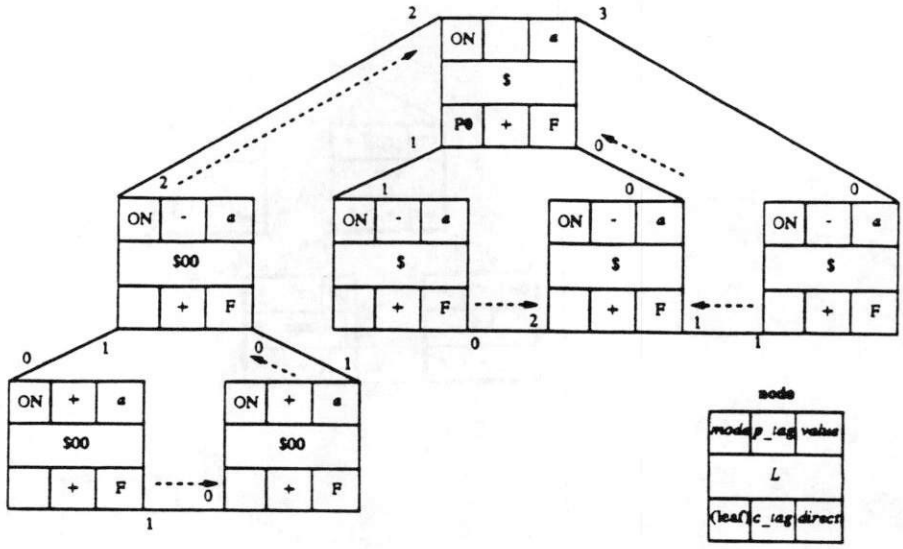

Figure 9: Initial configuration

DFS spanning tree generation procedure may produce loops (Figure 10a) due to unfortunate initialization of the labels. This causes overflows in the labels (Figure 10b) that guarantee eventual establishment of the minimally labeled tree (Figure 11d).

The configuration presented in Figure 11d is legally controlled and the *parent* fields in it induce the minimally labeled tree, although one of the nodes mistakenly regards itself as a leaf. Figure 11e presents the first legal configuration in the execution, where a solution is reached.

4 Network Consistency for Trees

In the rest of the paper we discuss protocols for a restricted class of network topologies — trees. Our aim is to see whether such a restricted class of problems can be solved using the more relaxed, uniform, distributed model, and whether it can result in a more efficient protocol.

It is well known that the sequential network consistency problem on trees is tractable, and can be achieved in linear time [16]. A special algorithm for this task is composed of an arc consistency phase (which is explained at the end of Section 3.4 and can be efficiently implemented on trees), followed by value assignment in an order created by some rooted tree. It has been shown that an arc consistent tree enables backtrack-free value assignment with no deadends [13]. Applying the general NC protocol together with the arc consistency protocol to a tree will already result in improved performance: when arc consistency is established, one forward phase of the value assignment protocol is sufficient to assign consistent values to all the nodes since no deadends occur (see also [8]). Therefore, the almost-uniform NC protocol if applied to trees is guaranteed to converge in a polynomial number of steps

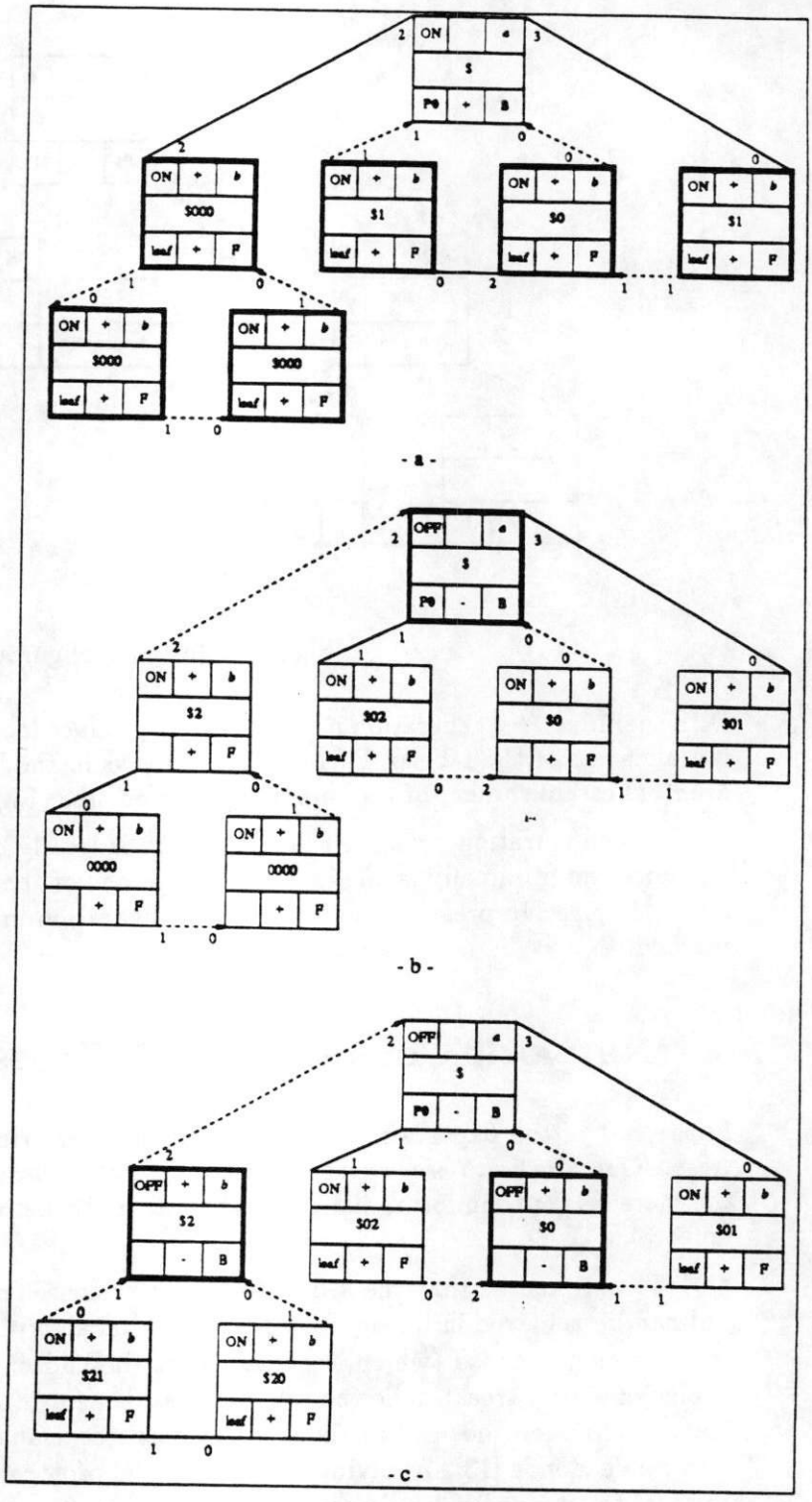

Figure 10: First steps to convergence

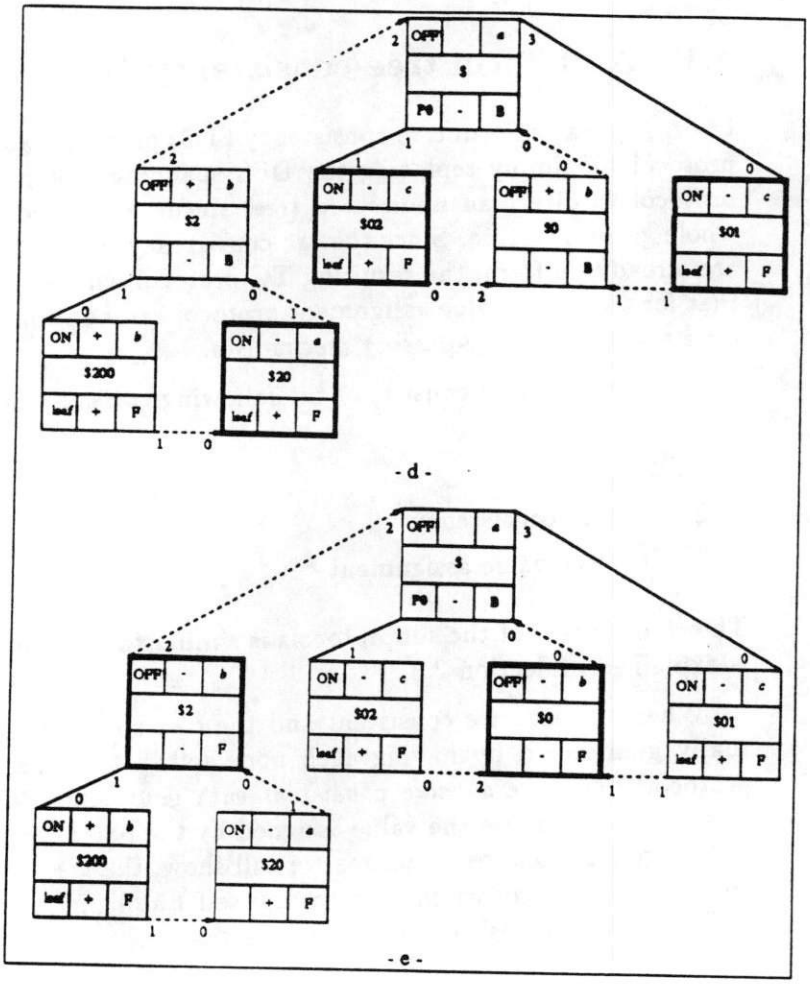

Figure 11: Network convergence

after the graph traversal mechanism has stabilized. There is a good chance for the privilege passing also to stabilize faster, since the arc consistency procedure decreases the domains.

Since the DFS spanning tree subprotocol of our general algorithm was the source for its non-uniformity, we reexamine the possibility that for trees, a rooted directed tree can be imposed via a uniform protocol. We have already shown that when using a distributed demon, a uniform, network-consistency protocol for trees is not feasible. Therefore, the only avenue not yet explored is whether under a central demon such a protocol does exist. We next show that this conjecture is indeed correct.

4.1 A uniform tree-consistency protocol

In principle a uniform tree-consistency (TC) protocol can be extracted from the general NC protocol by simply replacing the DFS spanning tree protocol with a uniform rooted-tree protocol to direct an undirected tree, since any rooted tree is also a DFS tree when the whole graph is a tree. Since the arc consistency protocol and the value assignment protocol are already uniform, the resulting TC protocol will be uniform. Nevertheless, we will show that for trees, the value assignment protocol can be simplified as well, while there is no need for a special privilege-passing mechanism.

The TC protocol consists of the following two subprotocols:

1. tree directing
2. (a) arc-consistency
   (b) tree value assignment

The composition of the subprotocols is similar to the general NC protocol (Section 3.2) and is explained in Section 2.3.

When the arcs are consistent and the tree has been directed, value assignment is eventually guaranteed by having each node follow the rule (of the **tree-value assignment** protocol): *"choose a value consistent with your parent's assignment"*. Such a value must exist, since otherwise the value assigned by the parent would have been removed by the arc consistency procedure. Since, as we will show, the tree directing protocol is self-stabilizing, and since the arc consistency protocol is self-stabilizing as well, the value assignment protocol eventually converges to a consistent solution.

4.1.1 Tree directing

In order to direct the tree uniformly, we must exploit the topology of the tree to break the symmetry reflected by the identical codes and the lack of identifiers. For this task we use a distributed protocol for finding the **centers** of a tree [15]. A center of a tree is a node whose maximal distance from the leaves is minimal. Consider a sequential algorithm that works in phases, so that in every phase the leaves of the previous phase are removed from the tree. In the last phase the tree has either one or two connected nodes left. These nodes are the centers of the tree.

Our protocol distributedly simulates the above algorithm. If only one center exists, it declares itself as a root (by setting its *root* field to TRUE), and all the incident edges are directed towards it. When two centers exist, one of them becomes a root and the link that connects them is directed accordingly. The choice of which center becomes the root is not deterministic and depends on the scheduling order and the initial values. The "first" center that applies the tree direction protocol declares itself as a root if and only if the other center is not the root already. When the other center is scheduled, it is supposed to do the same. However, its current status of either being the root or not is valid (since the first center has taken care of that already) and thus it remains unchanged. The central demon policy assures that only one (neighboring) center will be scheduled each time.

This approach yields a relatively simple uniform tree directing protocol that simulates the above description. Assuming the number of nodes in the network [2] is n, every node i has the following fields:

$N_i[0..\lfloor n/2 \rfloor]$ – a vector that counts the number of i's neighbors in each phase of the sequential algorithm. $N_i[j]$ records the number of neighbors of i in phase j. If $N_i[j] = 1$ it means that i becomes a leaf in the j-th phase (although it may be initialized incorrectly). $N_i[0]$ is repeatedly initialized to the number of i's neighbors in the network (so that $N_i[0] = 1$ means that i is a leaf in the original tree).

$root_i$ – a boolean field that indicates whether i is the root. Eventually only one of the centers has the value TRUE in this field.

$parent_i$ – a variable assigned the number of the edge that leads to the neighbor that becomes the parent of i (the enumeration is made by the α_i function, as in the general protocol, and is local to i). When the network stabilizes, namely when all the N-vectors converge, every node has only one neighbor that is eligible to be its parent, except the root which has none, and no two nodes are parents of each other.

The protocol works by having each node scan its neighbors' N-vectors and compute its own accordingly. The j-th entry of vector N_i represents the number of i's neighbors in the j-th phase of the sequential algorithm. Its value is recursively computed by decreasing the number of neighbors that became leaves in the previous phase from the entire number of neighbors in the previous phase, since those are exactly the neighbors of i that would have been removed by the sequential algorithm in its j-th phase. A node is recognized to be a leaf by having only one neighbor. Each node except the root chooses as its parent the neighbor that it is still connected to whenever it becomes a leaf, namely that neighbor which is not a leaf and therefore is not removed from the tree earlier than itself. All the entries of the N-vector after the one in which the node is recognized as a leaf are assigned a '/' value, indicating that the node is already removed from the tree.

A node recognizes that it is a center whenever one of the following two conditions is satisfied:

[2] We can overcome the necessity of knowing the size of the network by using dynamic memory allocation. However, for the sake of the simplicity of the code we assume the knowledge of n.

1. It becomes neighborless without being a leaf, which means that it is a single center of the tree and thus it becomes the root.

2. It becomes a leaf in the same phase as one of its neighbors — the other center. In this case, the node checks whether the other center is already the root. If-not, it becomes the root, and otherwise it chooses the other center to be its parent.

Figure 12 presents pseudo code for the protocol. Recall that the code is repeated forever, although from some point on, the tree does not change.

```
procedure tree-directing
Begin
1.   N_i[0] ← |neighbors(i)|
2.   for j = 1 to ⌊n/2⌋ do    { go over the N-vector }
3.        if N_i[j − 1] > 1 then          { if i is not yet a leaf at the (j − 1)-th phase }
          { the leaves of the (j − 1)-th phase are removed in the j-th phase }
4.             N_i[j] ← N_i[j − 1] − |{k | k∈neighbors(i) ∧ N_k[j − 1] = 1}|
5.             if N_i[j] = 0 ∨ (N_i[j] = 1 ∧ ∃k∈neighbors(i) s.t. (N_k[j] = 1 ∧ ¬root_k)) then
6.                  root_i ← TRUE              { i is the root }
7.                  parent_i ← NONE
8.             else                            { i becomes a leaf in the j-th phase }
9.                  parent_i ← k s.t. k∈neighbors(i) ∧ N_k[j] ≥ 1
                                    { eventually exactly one such k exists }
10.       else                  { i is not in the tree in the j-th phase }
11.            N_i[j] ← /
End.
```

Figure 12: Uniform tree directing procedure for node i

Proper convergence of the N-vectors is guaranteed by the fact that $N_i[j]$ depends only on $N_i[j-1]$ and $\{N_k[j-1] \mid k \in neighbors(i)\}$, which are properly updated earlier. The base of this iterative convergence is applied by repeatedly assigning to $N_i[0]$ the actual number of neighbors of i in the network.

The complexity of the tree protocol is clearly linear in the network's size since all its subprotocols are linear, and hence it equals the sequential time complexity. However, the parallel time can be further linearly bounded by the **diameter** of the tree where the diameter is the longest path between any two leaves of the tree.

4.2 Example

In order to demonstrate the convergence of the TC protocol, we present a simple tree constraint network and a typical execution. Figure 13 presents the initial configuration of the network and the structure of a node. The network is a tree with two centers. All the nodes

have identical domains $\{a, b, c\}$ and all the constraints, except those that are explicitly identified, are '=' (i.e., $\{(a,a),(b,b),(c,c)\}$). The *parent* field of every node is indicated as a direction of the appropriate edge. In the initial configuration all the nodes consider themselves as roots and have no parents. The values of the N-vectors at the beginning are random and meaningless.

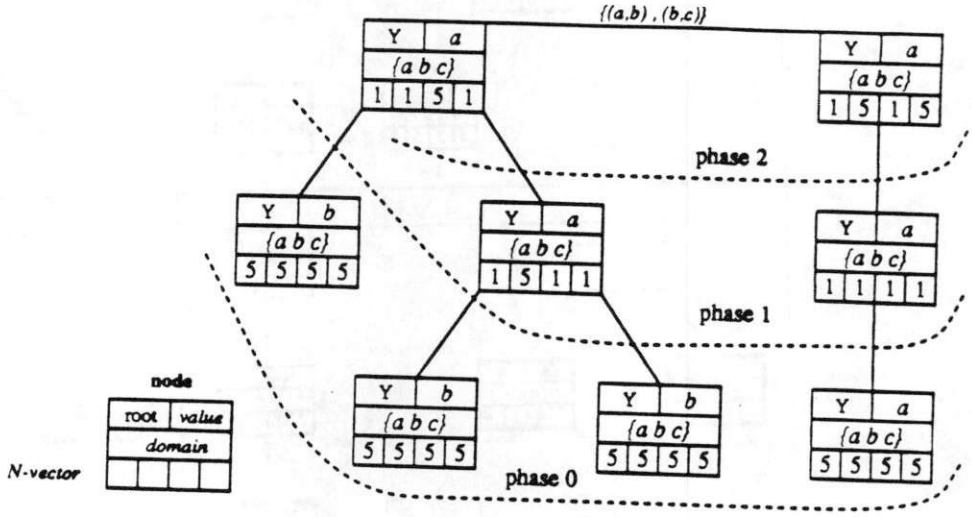

Figure 13: Initial configuration of tree network

Figures 13 and 14 present an execution prefix of five configurations until convergence to a solution. The configuration presented in 14d is legal and remains unchanged for the rest of the execution. The emphasized nodes in every configuration are those that were scheduled in the transition from the previous configuration to the current one, and thus their values might have changed.

Note how the converged N-vectors (Figure 14) correspond to the phases of the sequential center finding algorithm (indicated in Figure 13), and that eventually only one center nominates itself as the root. Note also that due to a bad choice of parents (14a) the assignment can temporally oscillate away from the solution (14a, 14b). However, eventually the right choice of parents is made (14c) and convergence to a solution is achieved (14d).

5 Conclusions

The results presented in this paper establish theoretical bounds on the capabilities of connectionist architectures and other distributed approaches to constraint satisfaction problems.

The paper focuses on the feasibility of solving the network consistency problem using self-stabilizing distributed protocols, namely, guaranteeing convergence to a consistent solution, if such exists, from any initial configuration. Such a property is essential for dynamic environments, where unexpected changes could occur in some of the constraints.

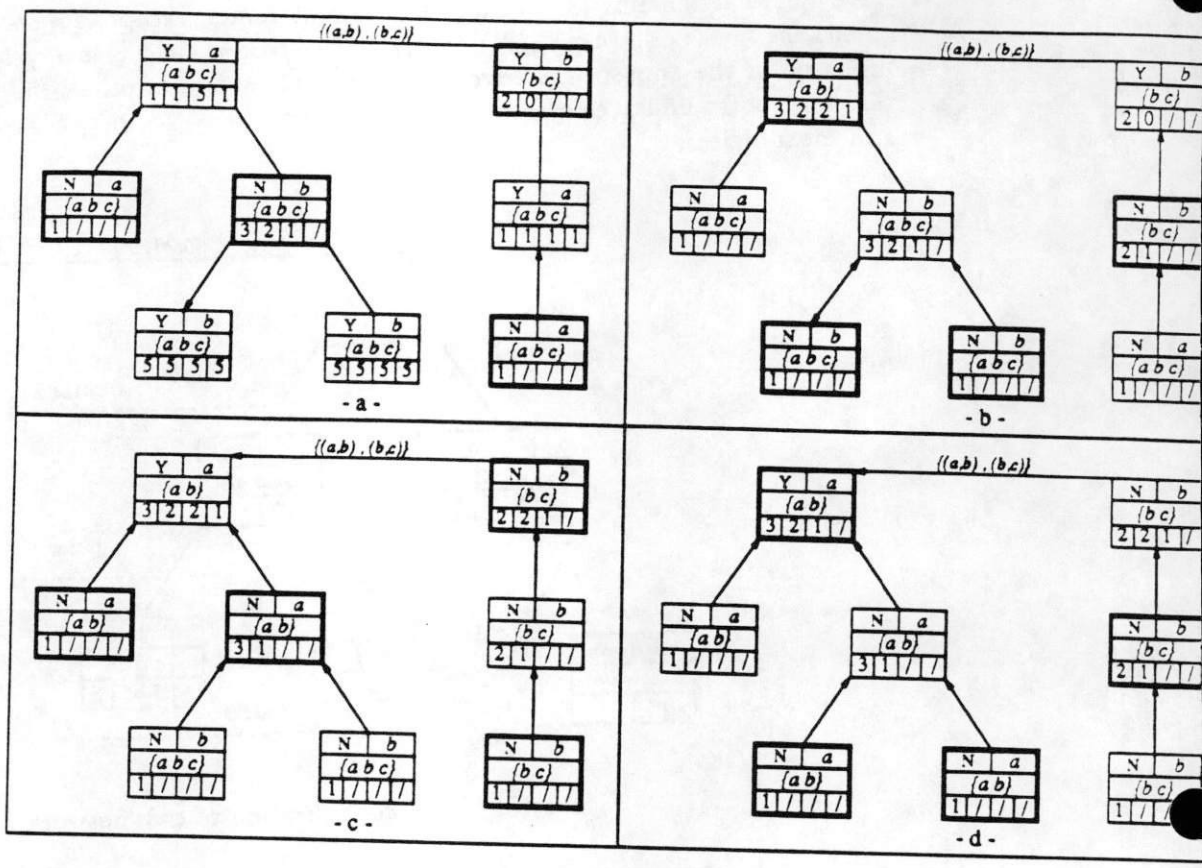

Figure 14: Tree network convergence

We proved that a uniform protocol (one in which all nodes are identical) when any schedule is possible cannot solve the network consistency problem even if only one node is activated at a time. Consequently, although such protocols have obvious advantages and are closer in spirit to neural network architectures, they cannot guarantee convergence to a solution. On the other hand, distinguishing one node from the others is sufficient to guarantee such a convergence even when sets of nodes are activated simultaneously. A protocol for solving the problem under such conditions is presented.

We then demonstrated that when the network is restricted to trees a uniform, self-stabilizing protocol for solving the problem for any schedule does exist, but only under a central demon (one neighboring node is activated at a time).

It is still an open question whether a uniform protocol is feasible for general graphs under some specific, given scheduling (e.g., round-robin).

Regarding time complexity, we have shown that in the worst case the distributed and the sequential protocols have the same complexity bound: exponential in the depth of the spanning DFS tree. On the average, however, a speedup is feasible by exploiting parallelism.

References

[1] D. H. Ballard, P.C. Gardner, and M. A. Srinivas. Graph problems and connectionist architectures. Technical Report 167, University of Rochester, Rochester, NY, March 1986.

[2] J. Burns, M. Gouda, and C. L. Wu. A self-stabilizing token system. In *Proceedings of the 20th Annual Intl. Conf. on System Sciences*, pages 218–223, Hawaii, 1987.

[3] Z. Collin and R. Dechter. A distributed solution to the network consistency problem. In *Proceedings of the 5-th Intl. Symp. on Methodologies for Intelligent Systems.*, pages 242–251, Tennessee, USA, 1990.

[4] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI-91*, Sydney, Australia, 1991.

[5] Z. Collin and S. Dolev. A self-stabilizing protocol for DFS spanning tree generation. in preparation, 1991.

[6] E. D. Dahl. Neural networks algorithms for an NP-complete problem: map and graph coloring. In *Proceedings of the IEEE first Internat. Conf. on Neural Networks*, pages 113–120, San Diego, 1987.

[7] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence Journal*, 41(3):273–312, January 1990.

[8] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88*, St. Paul, Minnesota, August 1988.

[9] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[10] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of PODC-90*, pages 103–118, Quebec City, August 1990.

[11] S. Even. *Graph Algorithms.* Computer Science Press, Maryland, USA, 1979.

[12] E. C. Freuder and M.J. Quinn. The use of lineal spanning trees to represent constraint satisfaction problems. Technical Report 87-41, University of New Hampshire, Durham, New Hampshire, 1987.

[13] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, January 1982.

[14] H.W. Gusgen and J. Hertzberg. Some fundamental properties of local constraint propagation. *Artificial Intelligence Journal*, 36(2):237–247, 1988.

[15] E. Korach, D. Rotem, and N.Santoro. Distributed algorithms for finding centers and medians in networks. *ACM Transactions on Programming Languages and Systems*, 6(3):380–401, July 1984.

[16] A. K. Mackworth and E.C. Freuder. The complexity of some polinomial network consistency algorithms for constraint satisfaction problem. *Artificial intelligence*, 25:65–74, 1985.

[17] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI-90*, Boston, 1990.